# Turtle Python 6 – Cellular Automata

This builds on the "Cellular Models" document, which introduced a lot of the key techniques. Cellular automata are cellular models whose dynamics – that is, the changes in patterning over time – are primarily determined by automatic, deterministic rules, which typically operate only "locally" and yet can give rise to interesting larger-scale patterns. The examples we shall be exploring are much harder to program than the previous cellular models, primarily because they require more storage of information as the changes are calculated. And so not only are the programs of interest in themselves, but they provide an excellent learning opportunity for a number of important and useful Computer Science concepts, concerned with the representation of numbers (in binary and hexadecimal) and "bitwise" operations on those numbers.

## 1. Introduction to Cellular Automata and Bitwise Pixel Manipulation

A *cellular automaton* is a grid of *cells*, typically in a rectangular array, each of which is in a particular *state* at any given moment. Initially, these states might be assigned randomly or in some pattern, but then as time "ticks" from moment to moment, the state of each cell may change, usually by following simple *rules* that determine the new state according to the arrangement of states across the neighbouring cells. Some cellular automata are *asynchronous*, with individual cells being processed one by one (as with the cellular models in the previous document). But the classic type of automaton is *synchronous*, with all cells being processed simultaneously, so that with each tick of the clock, we get a new *generation* of cell states, each of which has been individually determined by the same simple rules applied (within its neighbourhood) to the previous generation of states. These rules are standardly *deterministic*, so that the dynamics of the cellular automaton follow inevitably from the starting state. What makes these models so interesting is that even very simple deterministic rules can give rise to surprisingly elaborate patterns of development.

When implementing cellular automata within *Turtle* – just as with the examples in the "Cellular Models" document – it usually makes sense to change the resolution of the Canvas image (as well as the virtual Canvas dimensions) so that each cell corresponds to a single "pixel" (as well as a single coordinate location). Then the cell's colour can easily be set using the *pixset* command, and read using *pixcol*. Colours are represented by integers, stored in 3 "bytes" (i.e. 24 "bits"), corresponding respectively to the intensity of *red*, *green*, and *blue*, which are the three primary colours of light as far as human perception is concerned.[1] Hence such 24-bit integers are often called "RGB" codes.

RGB colour representation is far more fine-grained than ordinary human perception, and we cannot visually distinguish colours that are very slightly different. This provides a neat solution to a serious problem that arises when implementing *synchronous* cellular automata, in which cells across the entire grid have to be processed simultaneously, and which thus requires us to keep track *both* of each cell's state in the current generation, *and* its newly-calculated state in the next generation. As we shall see, we can do this using their RGB codes, but doing so requires being able to "pull apart" those codes and inspect the individual bits within them. This is where the opportunity arises to learn about binary and hexadecimal notation, bitwise operators, and using binary numbers and bitwise manipulation to encode rules and data.

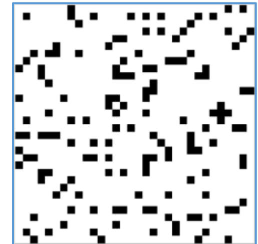
---

[1] Children are commonly told that the three primary colours are blue, red and yellow – often illustrated by mixing paints – but even in this context, it is at best an approximation. When mixing paint or ink, the primary colours are *cyan*, *magenta*, and *yellow*. *Cyan pigment* absorbs red light but reflects green and blue; *magenta pigment* absorbs green light but reflects red and blue; *yellow pigment* absorbs blue light but reflects red and green. So *a mixture of red and green light* gives a visual experience of *yellow*. But since red *pigment* absorbs green and blue light, while green *pigment* absorbs red and blue light, it follows that *a mixture of red and green pigment* will absorb all three RGB primary colours to a significant extent (and blue twice-over), thus reflecting a less bright mixture, visible as *brown* rather than *yellow*. The reason why red, green, and blue are primary colours is fundamentally due to the physiology of the human retina, which has three different kinds of "cone" cells, each particularly sensitive to light of one of those colours.

## 2. Introducing The Game of Life

The most famous cellular automaton, and one of the most fascinating, is John Conway's *Game of Life*. This involves a grid of square cells – potentially extending for ever in all directions – within which each cell can be either *alive* or *dead* (so there are just two possible cell states). Since an infinite grid is impractical, most computer implementations involve instead a finite square grid, e.g. 32×32, with the edges "wrapping around" (so the right-hand column is treated as being adjacent to the left-hand column, and the bottom row adjacent to the top row). Within this grid, live cells are usually shown black, and dead cells white – the initial arrangement of these black and white cells will often be set randomly, for example:

```
width = 32
height = 32
canvas(0,0,width,height)
resolution(width,height)
for x in range(width):
    for y in range(height):
        if randrange(7)==0:
            pixset(x,y,black)
```



Use of the two constants *width* and *height* – both here set to 32 – makes this code very easily adaptable for different sizes of board, just by setting the constants to different values. The *canvas* command then specifies the relevant coordinate range (i.e. 0 to 31 along both axes) and the *resolution* command fixes the corresponding image size (i.e. 32×32 pixels). Then the two variables *x* and *y* are used to count through all the cells in turn, randomly making roughly one in seven of them *black* and leaving the rest *white* (the function *randrange(7)* produces a random integer between 0 and 6 inclusive, so there is a 1 in 7 chance that it will be equal to 0).[2] And just as the pixels' colour can be set with the *pixset* command, so the *pixcol* function can be used to read that colour, e.g. `thiscol = pixcol(x,y)`. As already explained, *this makes it possible to use the pixel colour itself to record more detailed information about the state of each cell.*

Within the grid, we consider each cell as having 8 *neighbour* cells, as shown in the picture here. (This is a very common way of treating neighbourhoods in cellular models and automata, but some treat only cells 2, 4, 6, and 8 as neighbours, and other arrangements are also possible.) What happens to each cell then depends on whether it is alive or dead, and how many of its neighbours are alive or dead. In the case of the Game of Life, the rules are that from one generation to the next:

| 1 | 2 | 3 |
|---|---|---|
| 8 |   | 4 |
| 7 | 6 | 5 |

- A cell that is currently alive will stay alive in the next generation *if, and only if*, it currently has exactly 2 or 3 live neighbours. Otherwise, it dies.
- A cell that is currently dead will become alive in the next generation *if, and only if,* it currently has exactly 3 live neighbours. Otherwise, it stays dead.

Though simple, however, these rules are tricky to implement, precisely because they have to be applied to all cells simultaneously. For example, if *A* and *B* are neighbouring cells and we deal with *A* first, then if *A*'s state changes as a result, this risks messing up the calculation for *B*, whose next state should be determined (in part) by *A*'s *current* state, not by *A*'s *next* state. The upshot is that we need to be able to retain in memory *both* the current state of each cell, *and* the newly-calculated state that will take over in the next generation. One natural way of addressing this problem is to use *arrays* or *lists* to store the relevant information; this is illustrated in the *Turtle System* example program "Game of Life, using lists" (within Examples menu 7, "Cellular models"). But it is an interesting challenge to tackle the problem in a different way, by *using the*

[2] Note that the Canvas starts us as completely white, and any *resolution* command will create a new Canvas image in which every pixel is white. So we do not need here to set the colour of the individual white pixels.

*pixel colours to store both the current cell status and its future status*. This will also provide a vehicle for learning about a range of important new concepts.

## 3. Colour Codes in Binary and Hexadecimal

Taking this approach, then, there is no need to keep a separate record of whether each cell in the Game of Life is alive or dead – the pixels store that information already and can be manipulated individually using *pixset* and *pixcol*. And in fact the pixels can store far more information that this, because each pixel holds a three-byte *RGB* colour code, in which the *most significant byte* (i.e. the one written first, which has the biggest impact on the overall size of the hexadecimal number) represents the intensity of red, the middle byte the intensity of green, and the *least significant byte* the intensity of blue (this is exactly the same colour coding method that is used in web pages). Thus for example the RGB colour code for *emerald*, written out fully in binary but with commas between the individual bytes, is like this:

00000000, 11001001, 01010111

Each individual byte – consisting of 8 *binary digits* or *bits* (like 8 tiny electronic silicon switches, each of which can be either 0 "off" or 1 "on") – can take a value between 0 and 255, for example:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 00000000 | 0 | 00000110 | 6 | 00001100 | 12 | 11100000 | 224 |
| 00000001 | 1 | 00000111 | 7 | 00010000 | 16 | 11110000 | 240 |
| 00000010 | 2 | 00001000 | 8 | 00100000 | 32 | 11111000 | 248 |
| 00000011 | 3 | 00001001 | 9 | 01000000 | 64 | 11111100 | 252 |
| 00000100 | 4 | 00001010 | 10 | 10000000 | 128 | 11111110 | 254 |
| 00000101 | 5 | 00001011 | 11 | 11000000 | 192 | 11111111 | 255 |

The highlighted cases show the eight individual bits, whose values start with 1 at the right, then 2, then 4, then 8 and so on, doubling up each time as we move towards the most significant bit which represents 128.[3] Handling eight-digit binary numbers like these is obviously rather cumbersome, so for convenience it is standard to write them instead in *hexadecimal* notation, which is base 16 rather than 2. Consider, for example, the middle byte of the RGB code for *emerald*, as written above, but divided into two parts:

1100 1001

We can refer to each of these 4-bit parts as a *nybble* (since a nibble is a small bite!). The four binary digits of each nybble count for 8, 4, 2, and 1 respectively, so the binary number "1100" is equivalent to decimal "12" (8+4), and "1001" to decimal "9" (8+1). The highest possible value for a nybble is thus decimal "15" ("1111" = 8+4+2+1), so a nybble can take any value between 0 and 15, which enables it to be represented by a single *hexadecimal* digit (just as any value between 0 and 9 can be represented by a single *decimal* digit). Use of hexadecimal requires that we have 16 different digits available, so we go beyond "9" to use "A" as the hexadecimal digit for 10, "B" for 11, "C" for 12, "D" for 13, "E" for 14, and "F" for 15.

Consider again the code for *emerald*, again expressed in binary, but this time divided into six nybbles (rather than three bytes):

Binary: 0000 0000 1100 1001 0101 0111

If we now represent each individual nybble as a single hexadecimal digit, we get:

Hexadecimal (0x): 0 0 C 9 5 7

(Note that the binary number "1100" – 8+4 = 12 in decimal – corresponds to the hexadecimal digit "C".) We can now combine these nybbles in pairs to represent the entire colour code as three bytes again, but this time using hexadecimal rather than binary notation:

Binary: 00000000 11001001 01010111

[3] In our familiar decimal system, of course, these *place values* – starting from the right – are 1, 10, 100, 1000 etc., multiplying up by 10 each time. Thus the decimal number "11011" is equal to 1×10000 plus 1×1000 plus 1×10 plus 1, whereas the binary number "11011" is equal to 1×16 plus 1×8 plus 1×2 plus 1 (i.e. 27 in decimal).

| Hexadecimal (0x): | 00 | C9 | 57 |
|---|---|---|---|

So the hexadecimal colour code for *emerald*, as expressed in Python, is "0x00C957", with "0x" indicating that the number is *hexadecimal* (base 16), the red component being "0x00" (zero), the green component "0xC9" (12×16 + 9 = 201 in decimal), and the blue component "0x57" (5×16 + 7 = 87 in decimal). Since the maximum possible value for any component is 0xFF (15×16 + 15 = 255 in decimal), we can see that *emerald* is overall around 80% maximum intensity of green, mixed with around 33% blue (but no red at all).

## 4. Bitwise Operators

Once we understand binary and hexadecimal numbers, we can make use of *bitwise operators* to manipulate them. *Turtle* provides four standard operators, as follows:

| English name of bitwise operator: | *not* | *and* | *or* | *xor* |
|---|---|---|---|---|
| Python symbol for bitwise operator: | ~ | & | \| | ^ |

The *not* operator inverts the bits of the integer to which it is applied, taking it to have 32 bits altogether (so RGB codes are preceded by 8 binary zeros). Thus *not*(0), for example, will give #FFFFFFFF in hexadecimal, all 1-bits in binary.[4] The other three operators fix each bit in accordance with the following *truth-tables*:

> *These truth-tables fit with the standard "logical" interpretation of the three operators, taking 0 as false and 1 as true (thus "A and B" comes out true just when both A and B are individually true; "A or B" comes out false just when both A and B are individually false, and "A xor B" comes out true when A and B have different "truth-values", i.e. one of them is true and the other false).*

| | | *and* | *or* | *xor* |
|---|---|---|---|---|
| *A* | *B* | *A & B* | *A \| B* | *A ^ B* |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 |

So if, for example, we apply these operators between the decimal numbers 21 (binary 00010101) and 9 (binary 00001001), we get:

| | | | | | |
|---|---|---|---|---|---|
| | 00010101 (21) | | 00010101 (21) | | 00010101 (21) |
| | 00001001 (9) | | 00001001 (9) | | 00001001 (9) |
| &: | 00000001 (1) | \|: | 00011101 (29) | ^: | 00011100 (28) |

(21 & 9) yields 1, because the only 1-bits in the result are those that were 1-bits in both of the original numbers. (21 | 9) yields 29, because the only 0-bits in the result are those that were 0-bits in both of the original numbers. (21 ^ 9) yields 28, because the 1-bits in the result are those that were a 0-bit in one of the original numbers, and a 1-bit in the other.

Suppose now that we are given some six-digit hexadecimal colour code 0xRRGGBB and we want to get hold of the green component – the middle 8 bits (2 nybbles). We can do this by *anding* (i.e. using "&")

[4] In the "two's complement" system of integer representation (with 32 bits), #FFFFFFFF represents the integer -1. Hence because the number 0 is standardly taken to represent the truth-value *false*, it then becomes technically elegant to use -1 – i.e. *not*(*false*) – to represent *true*, thus enabling the very same operators to be used as "logical" connectives (connecting two statements) and "bitwise" operators (on two numbers). Following the standard languages from which they are derived, both *Turtle BASIC* and *Turtle Pascal* do exactly this, but *Turtle Python* – in accordance with standard *Python* – takes *true* to be equivalent to the number 1, and hence needs distinct logical and bitwise operators.

with the hexadecimal number 0x00FF00, which has all 1-bits in the middle byte and 0-bits elsewhere, and then integer dividing (using "//") by 0x100 (i.e. 256 in decimal).[5] Thus

| | |
|---|---|
| *emerald* & 0x00FF00 | evaluates to 0x00C900 (in hexadecimal) |
| (*emerald* & 0x00FF00) // 0x100 | evaluates to 0xC9 (in hexadecimal, 201 in decimal). |

If on the other hand we wish to add an element of red (say an intensity of 8/255) to *emerald* (0x00C957), then we can do this using the *or* (i.e. "|") operator:

| | |
|---|---|
| *emerald* \| 0x080000 | evaluates to 0x08C957 |

In this way, the *and* ("&") operator can be used to discover which bits are "set" (i.e. are 1-bits) in the binary representation of a number, and the *or* ("|") operator can be used to ensure that specific bits get set. The *xor* ("^") operator is useful when we want to change a particular bit (from 0 to 1, or 1 to 0).

## 5. Life and Death, Hiding in the Pixels

Let's now see how we can implement the Game of Life, storing all the intermediate information in the pixels. This provides a nice illustration of the techniques explained above, which can also be used for *steganography*, in which secrets are hidden in what look like ordinary pictures. (You can find the full program, entitled "Conway's Game of Life", in Examples menu 7, "Cellular models".)

So far, we have two colour codes in our Game of Life pixels, *black* and *white*. Spelled out in binary, with all 24 colour bits represented by a digit, *black* is "000000000000000000000000" and *white* is "111111111111111111111111". But it clearly makes life easier if we work in hexadecimal!

| | |
|---|---|
| *black* (live cells): | #000000 – zero intensity of red, green, and blue |
| *white* (dead cells): | #FFFFFF – maximum intensity of red, green, and blue |

*Every* bit in the code for *black* is 0, and *every* bit in the code for *white* is 1, but instead of using the entire number to indicate whether each cell is currently alive or dead, we could use just a single bit, for example the *least significant bit* (the very last, whose place value is 1). And then we could use the next least significant bit (the second last, whose place value is 2) to indicate something quite different, namely, *whether the cell is going to change state in the next generation*. So now we can add two new colour codes:

| | | |
|---|---|---|
| *blackish* (live but dying): | #000002 | (last byte is 00000010 in binary) |
| *whitish* (dead but resurrecting): | #FFFFFD | (last byte is 11111101 in binary) |

Visibly, *blackish* will be indistinguishable from *black*, and *whitish* will be indistinguishable from *white*. So we can store this information as we go along, without affecting how the Canvas looks! Changing a code from *black* to *blackish,* or *white* to *whitish* – or reversing such a change – is easily done by xoring the code with the number 2 (10 in binary), for example (*black xor* 2) = *blackish*, and (*blackish xor* 2) = *black*.

## 6. Applying the Game of Life Rules, with "Wrap-Around"

When calculating which cells are to die or be resurrected, we need to go through every cell in the grid, applying the rules we saw before:[6]

- A cell that is currently alive will stay alive in the next generation *if, and only if*, it currently has exactly 2 or 3 live neighbours. Otherwise, it dies.

[5] Most programming languages distinguish between standard division (symbol "/" – which in *Turtle* rounds to the nearest integer) and *integer division* (commonly called "*div*", with the symbol "//" in *Python*), which always rounds *down* to the nearest integer. Integer division is usually the more appropriate for bit-handling.

[6] Going through every cell in the grid involves a double loop of the form: `for x in range(width): for y in range(height):` (etc.). In the menu program, this occurs within a `while` loop whose condition – `?key!=\escape` – tests whether the *ESCAPE* key has been pressed (and terminates the loop if so). See §7 below for more on this.

- A cell that is currently dead will become alive in the next generation *if, and only if,* it currently has exactly 3 live neighbours. Otherwise, it stays dead.

Recall also that every cell has 8 neighbouring cells, even those on the edges of the grid, because the grid is supposed to "wrap around" from left to right, and from top to bottom. Perhaps surprisingly, this wrapping around is very straightforward, because we can take advantage of the *modulus* operator (written "%" in Python), which we met in the "Spirals and Shapes" document, and which yields *remainders* (so, for example, 18 % 7 evaluates to 4, because 7 goes into 18 twice, with a remainder of 4). If we are implementing a grid with a *width* of 32, so that the *x*-coordinates of our cells go from 0 to 31, then to find the "right-hand" neighbour of cell (*x*, *y*), instead of just adding 1 to the *x*-coordinate – which would get us to cell (*x*+1, *y*) – we include the *modulus* operator to get to cell ((*x*+1) % 32, *y*). Then in the crucial case where *x* is 31 (so we are already at the right-hand edge), we add 1 to get 32, but then find the remainder on division by 32, which yields 0 (i.e. 32 % 32 = 0). This makes it easy to "wrap around" from the right edge to the left edge.

Wrapping around from left to right is equally easy in Python, because of the way that the "%" operator works. So if we want to find the "left-hand" neighbour of cell (*x*, *y*), instead of just subtracting 1 from the *x*-coordinate – which would get us to the cell (*x*-1, *y*) – we again include the *modulus* operator to get to the cell ((*x*-1) % 32, *y*). Then in the crucial case where *x* is 0 (so we are already at the left-hand edge), we subtract 1 to get -1, but then find the remainder on division by 32, which yields 31 (i.e. -1 % 32 = 31).[7]

Thus if we're concerned with cell (*x*,*y*) and use variable *dn* to count "dead neighbours", then the Python code – now substituting *width* back in place of 32 – might go like this:

```
dn=0
for i in range(-1,2):
    for j in range(-1,2):
        dn = dn + pixcol((x+i)%width,(y+j)%height) & 1
```

Suppose for example, that *x* is 31 and *y* is 0 on a 32×32 grid, so the pixel (*x*,*y*) is at the top-right corner. That pixel is highlighted in the grid below. Then as *i* and *j* both count from -1 to 1, the expression "(*x*+*i*, *y*+*j*)" passes through 9 combinations of coordinates (including (31, 0) itself when both *i* and *j* are zero):

| | | | |
|---|---|---|---|
| (30, -1) | (31, -1) | (32, -1) | *j = -1; y+j = -1* |
| (30, 0) | (31, 0) | (32, 0) | *j = 0; y+j = 0* |
| (30, 1) | (31, 1) | (32, 1) | *j = 1; y+j = 1* |
| *i = -1; x+i = 30* | *i = 0; x+i = 31* | *i = 1; x+i = 32* | |

The shaded combinations are not legitimate gridpoints, because the coordinates in each direction run only from 0 to 31, so both -1 and 32 are "illegal" values. But suppose now that we do the following to the coordinates within each pair – *apply the "modulus 32" operator* (i.e. "%32"). The effect on the numbers shown above is as follows:

| *n* | -1 | 0 | 1 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|
| *n* % 32 | 31 | 0 | 1 | 30 | 31 | 0 |

Notice how -1 has "wrapped around" to 31, and 32 to 0, while the four legitimate values are unaffected. This changes our 9 combinations of coordinates to exactly what we want them to be:

[7] The Pascal *div* and *mod* operators (also BASIC DIV and MOD) work differently from the Python // and % operators when handling negative integers. If we imagine dividing *a* by *b* in the ordinary positive case, yielding a result of *q* (for integer *quotient*) and remainder *r*, then we have $a = (b \times q) + r$. This equation is generally respected in all languages, but when *a* is negative there is more than one plausible choice for *b* and *r*. In *Turtle Python* – as in standard Python – *q* is "floored" or *rounded towards negative infinity*, meaning that if *b* is positive then *r* (i.e. *a* % *b*) will always be in the range 0 .. *b*-1. In *Turtle Pascal* and *Turtle BASIC*, however, *q* will always be *rounded towards zero*; hence *r* can be negative when *a* is negative. (When *a* and *b* are both positive, of course, *rounding towards negative infinity* and *rounding towards zero* come to the same thing, which is why this difference emerges only with negative numbers.)

| | | |
|---|---|---|
| (30, 31) | (31, 31) | (0, 31) |
| (30, 0) | (31, 0) | (0, 0) |
| (30, 1) | (31, 1) | (0, 1) |

Thus our command:

```
dn = dn + pixcol((x+i)%width,(y+j)%height) & 1
```

does check the correct neighbourhood of pixels around (*x*,*y*), both at the edges and in the middle of the grid (and it's written in such a way that *width* and *height* could take values other than 32).

Now let's look at what's happening to *dn*. This is first set to 0, and then gets repeatedly incremented by (*pixel* & 1), where *pixel* is set equal in turn to the colour code of the 9 cells in the neighbourhood of (*x*,*y*) – i.e. cell (*x*,*y*) itself and its 8 neighbours. "(*pixel* & 1)" applies the bitwise *and* operator between *pixel* and #000001, thus yielding the value of the least significant bit, which is what we are using to record whether the cell is alive or dead (here we could equally well have used "(*pixel* % 2)", because the last bit is 1 if and only if *pixel* is odd, which means that the remainder on division by 2 will be 1). If the cell is alive, and *pixel* is *black* or *blackish*, then *pixel* is even, so (*pixel* & 1) will be 0. If dead, and *pixel* is *white* or *whitish*, then *pixel* is odd, so (*pixel* & 1) will be 1. Hence the command above does indeed succeed in counting the number of dead cells in the neighbourhood (and its doing so will be unaffected if some of the cells change colour from *black* to *blackish*, or from *white* to *whitish*). Having counted the number of dead cells, it's easy to modify the Game of Life rules accordingly:

- A cell that is currently alive will stay alive in the next generation *if, and only if*, there are currently 5 or 6 dead cells (i.e. 4 or 3 live cells) in the neighbourhood. Otherwise, it dies.
- A cell that is currently dead will become alive in the next generation *if, and only if,* there are currently 6 dead cells (i.e. 3 live cells) in the neighbourhood. Otherwise, it stays dead.

Applying these rules to cell (*x*,*y*) can now be done as follows using the bitwise *xor* operator, bearing in mind that (*black* ^ 2) is *blackish*, and (*white* ^ 2) is *whitish*. We want to make these colour changes (signifying an impending change of state) if *either* cell (*x*,*y*) is alive but the neighbourhood has neither 5 nor 6 dead cells, *or* if cell (*x*,*y*) is dead and the neighbourhood has exactly 6 dead cells:

```
if ((pixcol(x,y) & 1 == 0) and ((dn<5) or (dn>6)))
        or ((pixcol(x,y) & 1 == 1) and (dn==6)):
    pixset(x,y,pixcol(x,y)^2)
```

If we go through all the cells in the grid, performing these steps, then by the time we have finished, we will still have a grid that looks exactly the same (since *blackish* is indistinguishable from *black*, and *whitish* from *white*), but in which every cell that is due to change from live to dead, or vice-versa, will actually have subtly changed colour. And we will have done all this without allowing the calculations on each cell to affect those on its neighbours. All that remains, to finish this "generation" of the Game of Life, is to go through the entire grid updating *blackish* (i.e. dying) to *white* (i.e. dead), and *whitish* (i.e. resurrecting) to *black* (i.e. alive). This can be done routinely:

```
for x in range(width):
    for y in range(height):
        if pixcol(x,y) == 2:
            pixset(x,y,white)
        elif pixcol(x,y)=0xFFFFFD:
            pixset(x,y,black)
```

or more cleverly, with bitwise operators:

```
        if (pixcol(x,y) & 3) % 3 != 0:
            pixset(x, y, pixcol(x,y) ^ 0xFFFFFD)
```

This uses the fact that *blackish* differs from *white*, just as *whitish* differs from *black*, in every bit except for the second last. So bitwise *xoring* with #FFFFFD – which has every bit set *except for the second last* –

accomplishes exactly the changes we want. To test for the cells which need to be updated, we take the bottom two bits of the pixel colour (by bitwise *anding* with 3), and then find the remainder when this number is divided by 3. If the two bits are the same, then they make either 0 or 3 (00 or 11 in binary), hence zero remainder. If the two bits are different, we get a remainder of either 1 or 2, so we know that the current colour cannot be *black* or *white*; hence updating is needed. Having completed all these updates, we will be ready to start again on the next generation, and by putting the whole thing within a continuous loop (e.g. "while ?key<>\escape do", which will keep going until the ESCAPE key is pressed), we finish our implementation of the Game of Life. You can find this full program under "Conway's Game of Life" within Examples menu 7, "Cellular models".

## 7. Implementing the Game of Life with Lists

We have now seen how to implement the Game of Life while storing *within the pixel colours* all relevant information about the *current* and *future* state of the corresponding cells. This was concise and elegant, and has given a great opportunity to learn about colour representation and bitwise manipulation. But let us now look at a more conventional way of implementing this sort of model, using *lists* to store the *current state* and *future state* information quite separately from the digital image.

Here is the complete program "Game of Life, using lists" from Examples menu 7, "Cellular models", with comments appended:

```
width=32
height=32
canvas(0,0,width,height)
resolution(width,height)

thisgen=[]                          # create two-dimens THISGEN list
for x in range(width):              # WIDTH lists, each of size HEIGHT
    xlist=[]                        # create new XLIST as null list
    for y in range(height):
        xlist.append(0)             # every element is initially 0
    thisgen.append(xlist)           # append XLIST to THISGEN list

nextgen=[]                          # create two-dimens NEXTGEN list
for x in range(width):
    xlist=[]
    for y in range(height):         # this time, make element = 1
        xlist.append(randrange(7)==0)   # (i.e. True) with 1/7 chance
    nextgen.append(xlist)

while ?key!=\escape:                # until ESCAPE key is pressed ...
    noupdate()
    for x in range(width):
        for y in range(height):
            thisgen[x][y]=nextgen[x][y] # copy NEXTGEN into THISGEN
            if thisgen[x][y]:       # display THISGEN pattern
                pixset(x,y,maroon)
            else:
                pixset(x,y,lightgreen)
    update()                        # update Canvas with new pattern

    for x in range(width):          # for each cell [x][y] in THISGEN
        for y in range(height):
            livenb=0                # count live neighbourhood cells
```

```
            for i in range(-1,2):
                for j in range(-1,2):
                    if thisgen[(x+i+width)%width][(y+j+height)%height]:
                        livenb+=1

            if thisgen[x][y]:           # apply rules to create NEXTGEN
                nextgen[x][y]=((livenb==3) or (livenb==4))
            else:
                nextgen[x][y]=(livenb==3)
```

For variety, this program uses a variable *livenb* to count the number of *live* neighbours (rather than variable *dn* which counted the number of *dead* neighbours), and it uses different colours (*maroon* and *lightgreen*). But much more importantly, note the use of *two-dimensional lists* to store information about both the *current* generation of cells (*thisgen*) and the *next* generation of cells (*nextgen*). *thisgen* is initially created as an empty (or null) list:

```
thisgen=[]
```

Then, moving from left to right across the grid, for each *x*-coordinate (i.e. each column of cells), an *xlist* is created which contains a cell for every *y*-coordinate in that column:[8]

```
for x in range(width):
    xlist = []
```

So just like *thisgen*, the *xlist* starts out empty.

```
    for y in range(height):
        xlist.append(0)
```

Now 0 (the integer) has been appended to the *xlist*, *height* times. So if *height* were 5 (say), *xlist* would now have become [0, 0, 0, 0, 0].

```
    thisgen.append(xlist)
```

this appends the newly created *xlist* to *thisgen*, so each time round the loop – for *width* iterations altogether – *thisgen* gains an extra sublist. Thus if *width* is 4 and *height* is 5 (say), *thisgen* progressively grows like this:

[]     initial null list

[[0, 0, 0, 0, 0]]     after 1 iteration

[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]     after 2 iterations

[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0] , [0, 0, 0, 0, 0]]     after 3 iterations

[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]     after 4 iterations

*nextgen* is created in the same way, except that instead of (in effect) inserting 0 into every cell, the loop:

```
    for y in range(height):
        xlist.append(randrange(7)==0)
```

first chooses a random number between 0 and 6 (inclusive), and then appends either `True` or `False` depending on whether or not that random value is equal to 0. In Python, `True` is equivalent to 1, and `False` is equivalent to 0, so this loop this has the effect of inserting 0 into each cell with probability 6/7, and 1 into each cell with probability 1/7. This generates the initial random pattern of living and dead cells. Having

[8] It's worth noting for future reference that in standard Python, a new assignment to `xlist` *creates a new list*, rather than simply changing the value of the existing `xlist` variable. If this were not the case, then in repeatedly appending `xlist` to `thisgen`, we would simply be appending numerous copies of the very same list, which would mean that every sublist in `thisgen` would be – and remain – identical. Avoiding this requires creation of a new `xlist` each time before appending – we cannot just create our initial `xlist` and then repeatedly append it to `thisgen`. (*Turtle Python* actually handles all this slightly differently, because of the way it is *compiled*, but we can ignore the details here.)

done this, the program enters its main generational loop, which takes the overall form:

```
while ?key != \escape:
    1. Copy nextgen list into thisgen list and display thisgen pattern
    2. Create nextgen list from thisgen list using LIFE rules
```

The `while` statement allows the loop to continue repeatedly until the *ESCAPE* key is pressed. Details of handling keyboard input are in the "Animation and User Input" document, but briefly, the statement uses `?key` to pick up the code of whatever key may have been last pressed, and `\escape` is a symbolic name for the *keycode* of the *ESCAPE* key (which is actually 27). So when *ESCAPE* is pressed, the loop exits.

Within the main generational loop, the first stage is to copy the *nextgen* list into *thisgen* and to colour all the cells (maroon for *live*, and lightgreen for *dead*) accordingly. This is all enclosed between the commands `noupdate()` and `update()`, to ensure that the screen updating occurs quickly and smoothly over the entire grid, after all the pixel setting instructions have been issued:

```
noupdate()
for x in range(width):
    for y in range(height):
        thisgen[x][y] = nextgen[x][y]  # copy NEXTGEN into THISGEN
        if thisgen[x][y]:              # display THISGEN pattern
            pixset(x,y,maroon)
        else:
            pixset(x,y,lightgreen)
update()                             # update Canvas with new pattern
```

Here it's important to avoid a tempting mistake. Notice that here we are copying each individual element of the *nextgen* list into the *thisgen* list, using the instruction `thisgen[x][y]=nextgen[x][y]`. But wouldn't it be simpler instead just to copy the entire list before starting on the FOR loop, as follows?

```
thisgen=nextgen
for x in range(width):
    for y in range(height):
        if thisgen[x][y]: (etc.)
```

But this would not work, because the instruction `thisgen=nextgen` doesn't merely *copy the contents of one list to the other – it makes thisgen refer to the very same list as nextgen*. So *thisgen* just becomes another name for *nextgen*, and the distinction between the generations falls apart, bringing odd results.

Finally, the rules of the Game of Life, transforming from *thisgen* to *nextgen*, are applied through the following commands, looping through each cell (*x*,*y*) on the Canvas:

```
for x in range(width):               # for each cell [x][y] in THISGEN
    for y in range(height):
        livenb=0                     # count live neighbourhood cells
        for i in range(-1,2):
            for j in range(-1,2):
                if thisgen[(x+i)%width][(y+j)%height]:
                    livenb+=1
        if thisgen[x][y]:            # apply rules to create NEXTGEN
            nextgen[x][y]=((livenb==3) or (livenb==4))
        else:
            nextgen[x][y]=(livenb==3)
```

Here *livenb* is initially set to 0 – this will count the number of *live* neighbours of the cell (*x*,*y*). Then, as in the previous version of the program, we have nested counts of *i* and *j* from -1 to +1, wrapping around at the edges of the Canvas, so that all nine cells in the appropriate neighbourhood are progressively identified by the coordinates "(*x*+*i*) % *width*" and "(*y*+*j*) % *height*". If that element of the *thisgen* array is *True* (i.e.

value 1) – indicating that the relevant neighbouring cell is *alive* in the current generation – then *livenb* is accordingly incremented, so that by the end of these nested counts, *livenb* represents *the number of living cells in the neighbourhood of nine cells, including the central cell (x,y)*. The final if … else … conditional instruction interprets the standard Game of Life progression rule: if cell (*x*,*y*) is currently alive (i.e. `thisgen[x,y]` has the value 1, i.e. `True`), then it stays alive in the next generation if (and only if) the total number alive in the neighbourhood *(including itself)* is 3 or 4. If it is currently dead, then it becomes alive if (and only if) the total number alive in the neighbourhood is exactly 3. Once this has been done for every cell on the Canvas, the *nextgen* array represents the live/dead status of every cell in the next generation. So now the program loops back to update *thisgen* from *nextgen*, and we repeat the generational loop.

## 8. Investigating One-Dimensional Cellular Automata

The simplest cellular automata have two possible states per cell (like the Game of Life) but are only one-dimensional, meaning that all the activity takes place along a single line. Though simple, however, this brings a new element of interest for us, because it enables us to picture what happens through the generations, by showing them on successive lines of the Canvas. As usual, we'll fix the states of the first generation of cells (along the top line of the Canvas) randomly.

Suppose we consider the states as 0 and 1 – showing as white and black respectively (though we'll use the visually indistinguishable #FFFFFE for white and #000001 for black, so that the bottom bit of each colour code yields 0 and 1 respectively). If we take the *neighbourhood* of each cell to consist of itself and the two adjacent cells, then there are eight possibilities for each neighbourhood:

111 110 101 100 011 010 001 000

Depending on the nature of our automaton's *transition rule* – which is assumed to be *deterministic* (i.e. not chancy) – each of these possibilities will lead to one of two specific outcomes in the next generation, making the cell in question (i.e. the middle one of the three) either 0 or 1. For example, our rule might specify the following transitions for the eight possible neighbourhood situations:

| | 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
|---|---|---|---|---|---|---|---|---|
| | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| (R) | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

Suppose we apply this rule to a first line that has randomly turned out like this:

| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|

The rule depends on 3-cell neighbourhoods, but the two end cells don't have such a neighbourhood (because they have only one neighbour); hence we imagine the line "wrapping around" so that the "0" at the right-hand end is considered as adjacent (on the left) to the "1" at the left-hand end:

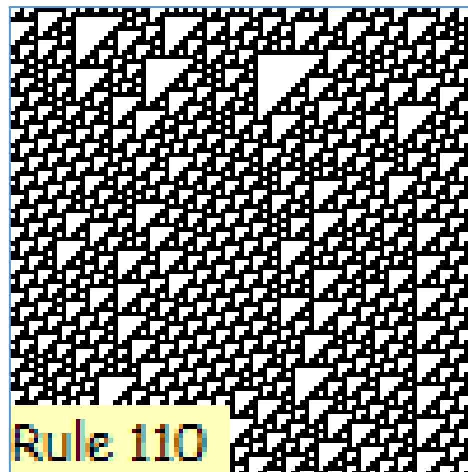
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Then the result for each of the cells, derived by examining its 3-cell neighbourhood and consulting the relevant part of the rule, will be:

| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|

Thus in the first cell, we see "011" leading to "1"; in the second, we see "111" leading to "0"; in the third, we see "110" leading to "1"; in the fourth, we see ""101" leading to "1", and so on.

Of course this is only one possible transition rule, and since there are two possible transitions for each of eight possible neighbourhood situations, it follows that there are $2^8$ – i.e. 256 – possible rules altogether. Specifying these rules is very simple, because each transition is to either "0" or "1", so each such choice can be treated as a binary digit, giving us rules numbered from "0000000" (zero) to "11111111" (255 in decimal).




And we can see from (R) above that by this method, the illustrated rule is number "01101110" in binary, or 110 in decimal. The pattern produced by this rule is shown here, as displayed (along with many others) by the program "One-dimensional cellular automata", which can be found within Examples menu 7, "Cellular models". This counts through an interesting subset of these 256 rules, and it can easily be modified to count through all of them.

The program starts with the usual specification of *width* and *height* (here both are 100), then defines the two-element list *cellcol* so as to contain appropriate colour codes (#FFFFFE and #1 – almost *whit*e and almost *black* respectively) for *dead* and *live* cells respectively. This neatly gives us that *cellcol[0]* is the right colour for any cell whose value is 0, and *cellcol[1]* is the right colour for any cell whose value is 1, so the binary digit resulting from any transition can easily be translated into a pixel colour. An 8-element list *nextstate* (initially all zeros) is also defined, to be used by the *setup* and *nextgen* functions.

Aften setting up the Canvas and resolution in the usual way, the main program consists of a large *for* loop which sets the variable *n* counting from 4 to 45, while in turn *rule* is made equal to (4*n*+2), so that *rule* counts from 18 up to 182 in steps of 4 (these numbers just happen to give an interesting subset of rules). Each time round the loop, the *setup* routine is called to work out the details of the rule in question, the Canvas is cleared to white, and the cells in the top line – generation 0 – are filled randomly with either *cellcol[0]* or *cellcol[1]*. The rest is then filled in by repeated calls of the *nextgen* procedure, implementing the generations from 1 to the bottom of the Canvas (so the number of the final generation is *height-1*).

The *setup* routine takes a rule number (as explained above) as a parameter, and identifies each of its binary digits in turn (by finding the remainder on division by 2, then executing rounded-down integer division by 2 and continuing). Each of these digits is stored in the *nextstate* array, so *nextstate[5]*, for example, will then specify the required transition for the neighbourhood "101" (which is 5 in binary).

The *nextgen* routine takes the generation number, *g*, as a parameter, and counts through the pixels on Canvas row *g-1* (i.e. the parent row), working out which transitions should be applied to create the pixels on row *g*. Although the parent pixels are indexed horizontally from 0 to *width*-1, this count is extended at each end, going from -1 to *width*, to capture all the 3-cell neighbourhoods using the remainder (or *modulus*) operator % in something like the now familiar way:

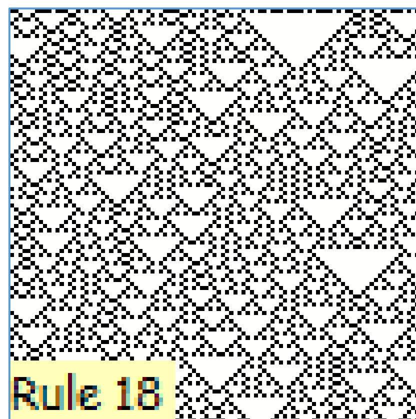
```
for x in range(-1,width+1)
    thispix=pixcol(x%width,g-1) & 1
    n3=n2*2+thispix
    n2=n1*2+thispix
    n1=thispix
    if x>0:
        pixset(x-1,g,cellcol[nextstate[n3]])
```

At the extremes, when *x* is -1, *x%width* will be (*width*-1), and when *x* is *width*, *x%width* will be 0; in all other cases, *x* and *x%width* are equal. This means that as *x* counts from -1 to *width*, *x%width* will count from (*width*-1) to 0, wrapping round from the right-hand edge of the Canvas to the left, and thus including the full neighbourhood of both extreme cells. As this count proceeds, *thispix* becomes either 0 or 1, depending on the pixel value of the relevant cell (with "& 1" ensuring that we take only the last binary digit – as we saw with the Game of Life, "*%* 2" would have exactly the same effect). Then, treating the next three lines in reverse, *n1* is made equal to *thispix*, *n2* is made equal to *thispix* plus twice the *previous* value of *n1*, and *n3* is made equal to *thispix* plus twice the *previous* value of *n2*. So after we have gone round this loop three

times (with *x* equal in turn to -1, then 0, then 1) – and using *p[n]* here to signify the value (i.e. 0 or 1) of the $n^{th}$ pixel on row (*g*-1) – these variables will be set as follows:

*x*: 1    *n1*: *p*[1]    *n2*: *p*[0]×2 + *p*[1]    *n3*: *p*[*width*-1]×4 + *p*[0]×2 + *p*[1]

Thus *n3* will now have a value which, in binary, reflects the neighbourhood of cell 0 in generation (*g*-1). And so to apply the necessary transition rule, we make the next-generation cell at (0, *g*) equal to *nextstate*[*n3*], and set the corresponding pixel colour to *cellcolour*[*nextstate*[*n3*]]. Meanwhile, *n2* and *n1* are primed to go round the loop again, so as to recalculate *n3* to give the neighbourhood of cell 1 next time (when *x*=2), and so on all the way to cell (*width*-1), whose neighbourhood transition will be applied when *x* finally reaches *width* (at which point *xmod* will have wrapped round to 0).




Following the calculation of the remaining generations, the program ends by displaying the rule number, at the bottom left of the Canvas:

```
setxy(0,height-15);
box(25+length(str(rule))*7,14,cream,False);
print('Rule '+str(rule),4,8);
```

This first line moves the *Turtle* to location (0,85), assuming the *height* is 100. Then it draws a cream-coloured "box" (without a border), whose height is 14 pixels and whose width is:

25+length(str(rule))*7.

Depending on whether the rule number has 1, 2 or 3 digits, this gives a box width of 32, 39, or 46 respectively. Then we print "Rule 110" (or whatever) in *Turtle* font number 4 (Comic Sans) and size 8. As can be seen in the image of rule 18 above, this gives as neat a finish with a two-digit number as we saw before with three.

## 8.1 Cellular Automata, Patterns in Nature, and Alan Turing

If we compare the pattern produced by rule 18 with the pattern on the shell of a *Conus textile* sea snail (pictured here), one cannot but be struck by their amazing similarity. This seems unlikely to be coincidence, especially when we consider that the snail's shell has been built up through growth along the line of its edge, so that its pattern is the result of a sequence of lines generated over time, just like our cellular automaton patterns.

In 1952, Alan Turing published a paper on "The Chemical Basis of Morphogenesis", in which he hypothesised a "reaction/diffusion" process for the creation of biological patterns such as on the coats of zebras and leopards. In recent years, this theory has inspired cellular automaton models that work in a functionally similar way, and the use of such models to investigate the formation of patterns in nature has become a significant area of fascinating research.

# 9. Implementing the Iterated Prisoner's Dilemma

*Evolutionary game theory* involves analysing the outcome of behaviour – whether by humans, animals, or other living things – in terms of the "payoff" to each party, and provides a valuable method of modelling such phenomena. One of the central questions of evolutionary biology concerns the emergence of *altruism*, whereby individuals apparently act in *cooperative* ways that assist their fellows, even at cost to themselves, and even when they are not closely related biologically. This sort of non-family cooperation is much harder to explain than cooperation within families, because the latter makes sense in terms of "kin selection", whereby a gene that leads individuals to favour family members will flourish, precisely because that gene

is likely to be possessed by those other family members who benefit from such unselfish behaviour. (This fits well with the so-called "selfish gene" perspective popularised by Richard Dawkins, whereby we can be seen as *gene vehicles* that are being manipulated by our genes to aid their own successful propagation.) From now on, we focus on situations where the individuals concerned are *unrelated* to each other.

## 9.1 Introducing the Prisoner's Dilemma

Selfishness and altruism are often modelled in terms of the famous "Prisoner's Dilemma", where two individuals each have to choose – *independently and without knowledge of the other's choice* – whether to "cooperate" with the other or to "defect". The characteristic feature of Prisoner's Dilemma type situations is that the potential benefits and costs give a clear incentive for each individual to "defect" on the other one, irrespective of what choice the other makes. But despite this, *both* individuals do better if they both choose to "cooperate".[9] It is a "dilemma", because individual rationality seems to lead both of them into a sub-optimal situation, whereas naïve good will could have left them both better off. How is it possible that *rational* choice of action with a view to maximising one's own benefit can be sub-optimal in this way?

Analysis of the Prisoner's Dilemma is standardly done in terms of some sort of "payoff matrix", and the following such matrix (in terms of the proportions between the various payoffs) is the most common. Here the payoffs are expressed in terms of my own point of view, but they are symmetrical – both individuals face exactly the same decision, with the same potential payoffs:

| | **You cooperate** | **You defect** |
|---|---|---|
| **I cooperate** | $**3,000** *(Reward)* | $**0** *(Sucker's Payoff)* |
| **I defect** | $**5,000** *(Temptation)* | $**1,000** *(Punishment)* |

In these circumstances, it seems eminently rational for me to reason like this:

> "If my 'partner' cooperates, then I can get a payoff of $5,000 by defecting, against only $3,000 by cooperating – so in that case, I'm better off defecting."
>
> "If my 'partner' defects, then I'll get nothing at all if I cooperate, and at least defecting will get me $1,000 – so in that case too, I'm better off defecting."

But if both of us reason in this "eminently rational" way, we both end up with just $1,000, whereas if we had "naïvely" cooperated in order to be nice to the other, we'd have both got $3,000!

## 9.2 Evolution and Iteration

In the *Prisoner's Dilemma* (PD), as we've seen, the parties act quite separately, and the only self-interestedly "rational" action seems to be to defect. It seems plausible to conclude that *evolution* of a species which frequently operates within this sort of scenario could only favour defection behaviour. But in nature – and especially amongst humans – a huge amount of cooperation does seem to occur. How might this have evolved, when individual benefit always seems to be favoured by defecting?

American political scientist Robert Axelrod realised that the answer might be found by considering *patterns of repeated behaviour* rather than individual choices. So if we imagine individuals interacting

[9] In the original "Prisoner's Dilemma" story, two criminals are being separately interrogated for a crime that they did in fact commit together, but where the police have insufficient evidence to convict them without a confession from at least one of them. Here "cooperating" with the other criminal involves keeping silent, while "defecting" is owning up to the police in the hope of being let off as a reward for incriminating the other. If both "cooperate" (with each other), then they will be punished only for some lesser crime, whereas if both "defect" (on each other), then both will be convicted of the more serious crime, albeit with a reduced publishment for having confessed. The downsides of this famous example are first, that "cooperate" here looks ambiguous (it might be misinterpreted as meaning cooperation *with the police*), and secondly, that the "payoffs" are negative (in terms of time in jail, rather than positive benefit).

*repeatedly* with each other – for example in what is called an *Iterated Prisoner's Dilemma* (IPD) – and potentially *adapting* their behaviour on the basis of what they *learn* from the experience of past encounters, then this could provide a plausible means of explaining how cooperative behaviour can arise and thrive. But investigating *extended patterns* of behaviour is potentially very hard, because there are so many possibilities. In the "one-shot" Prisoner's Dilemma there is a simple choice between just two courses of action, but when it is iterated, there could be a host of potential strategies to consider.

As a way of dealing with this complexity, in the early 1980s Axelrod had the brilliant idea of arranging a series of computer "IPD tournaments", in which he invited a varied range of academics to compete *by submitting computer programs that implemented different IPD strategies*. These programs were then made to interact together in a series of all-play-all IPDs, and their payoffs from the encounters were tallied to see which programs performed best. Note that this sort of competitive scenario also lends itself to being modelled evolutionarily – for example, we might imagine a population of individuals, each following a particular IPD strategy and interacting repeatedly (say, 10 times) with each other member of the population. Then after each "generation" of such interactions, the most successful strategies (i.e. those that have acquired the highest aggregate payoff) could "multiply" within the contest environment, so their frequency grows within the population (while the least successful reduce in frequency or are lost entirely). Over time, we might see an interesting evolutionary pattern in the relative frequencies of the strategies.

There is potentially a vast number of possible strategies in the Iterated Prisoner's Dilemma, but here are some straightforward examples that are very well known, with their standard names and codes:

- ALLD Always Defect
- ALLC Always Cooperate
- ALTCD Alternately Cooperate then Defect
- TFT Tit-for-Tat: start by Cooperating, then respond in kind to partner's last action
- TF2T Tit-for-Two-Tats: Defect only immediately after partner has defected twice in a row
- GRIM Cooperate until partner first defects, but continuously Defect from then onwards

It is obvious that some of these strategies are "nicer" than others. For example, ALLC, TFT, TF2F and GRIM will *never* defect until their partner first does so. At that point, TFT and GRIM would immediately retaliate, but TF2F is more forgiving, avoiding retaliation if the partner then immediately goes back to cooperating. TFT is also forgiving whenever the partner resumes cooperation, but GRIM is completely unforgiving: once provoked, it will *never* go back to cooperating. If we array the strategies in order of increasing "niceness", and work out how they would do against each other in an IPD with 10 interactions (and the standard payoffs of 3 for mutual cooperation, 1 for mutual defection, 5 as "temptation" and 0 "sucker"), we get this table:

| | **ALLD** | **ALTCD** | **GRIM** | **TFT** | **TF2T** | **ALLC** |
|---|---|---|---|---|---|---|
| **ALLD** | 10/10 | 30/5 | 14/9 | 14/9 | 18/8 | 50/0 |
| **ALTCD** | 5/30 | 20/20 | 12/27 | 28/23 | 40/15 | 40/15 |
| **GRIM** | 9/14 | 27/12 | 30/30 | 30/30 | 30/30 | 30/30 |
| **TFT** | 9/14 | 23/28 | 30/30 | 30/30 | 30/30 | 30/30 |
| **TF2T** | 8/18 | 15/40 | 30/30 | 30/30 | 30/30 | 30/30 |
| **ALLC** | 0/50 | 15/40 | 30/30 | 30/30 | 30/30 | 30/30 |

Here a red cell indicates that the *row* strategy is vastly outperforming the *column* strategy, with pink indicating a smaller disparity. Likewise a blue cell indicates that the *column* strategy is vastly outperforming the *row* strategy, with cyan indicating a smaller disparity. The green cells show both strategies performing equally well, and the large block of bright green cells are those that represent a fully cooperative series of interactions, with both parties scoring 3 on all 10 iterations.

## 9.3 The Iterated Prisoner's Dilemma Example Program

The "Iterated Prisoner's Dilemma" program in the menu "Examples 7 – cellular models" focuses on a relatively simple competition between just three strategies:

- ALLC Always Cooperate – initially in the great majority (84%), but exploited by ALLD
- ALLD Always Defect – initially only 2% of the population, but grow quickly
- TFT Tit-for-Tat – initially 14% of the population – far more resistant that ALLC to ALLD

This program is intended as an illustration of the interesting dynamics that can occur in such a competition, and of some of the techniques that can be used to implement what is quite a complex task. The program is quite short, but nevertheless conceptually tricky, because (as we saw in the Game of LIfe) it neatly uses bitwise processing to enable the necessary intermediate results to be stored within the Canvas pixels.

## 9.4 Initial Canvas setup and Program Main Loop

The program starts by defining three constants:

```
width=40    # width of the Canvas
height=40   # height of the Canvas
n=10        # number of iterations of PD between neighbours
```

Moving down now to the main program (after the three function definitions), we can see how the Canvas is initially set up:

```
# Main program, starts by setting Canvas
canvas(0, 0, width, height)
resolution(width, height)

# Set original map, with 2% blue, 14% green and 84% red
noupdate()
for i in range(width):
    for j in range(height):
        if randrange(50)==0:
            pixset(i, j, 0x0000FC)
        elif randrange(7)==0:
            pixset(i, j, 0x00FF02)
        else:
            pixset(i, j, 0xFF0001)
update()

# Pause to show original map
pause(1000)
```

As the "Set original map" comment suggests, this populates the Canvas as follows, and here "#" is used to indicate an RGB colour code:

| | |
|---|---|
| 2% (1 in 50, randomly): | blue = #0000FC |
| 14% (1 in 7 of the remaining 98%): | green = #00FF02 |
| 84% (the remainder): | red = #FF0001 |

But notice that the RGB codes here are _not_ quite the standard codes for *blue*, *green*, and *red* (which would be #0000FF, #00FF00 and #FF000 respectively). Instead, if we set out the codes in binary, we can see that the last three bits of each colour code are being set so as to indicate the type of each cell

| | | | | |
|---|---|---|---|---|
| blue | = | 0000 0000 | 0000 0000 | 1111 1100 |
| green | = | 0000 0000 | 1111 1111 | 0000 0010 |
| red | = | 1111 1111 | 0000 0000 | 0000 0001 |

So the bottom four bits are 1100 for blue, 0010 for green, and 0001 for red. These bottom four bits will be preserved even while the other parts of the pixel colours are being adjusted in the subsequent processing. Now we move on to the main loop of the program, which continues until either the *escape* key or the "HALT" button is pressed :

```
# Repeatedly update generations until Escape is pressed
while ?key != abs(\escape):
    noupdate()
    for i in range(width):
        for j in range(height):
            utility(i, j)
    for i in range(width):
        for j in range(height):
            pickbest(i, j)
    for i in range(width):
        for j in range(height):
            fixbest(i, j)
    update()
```

This is straightforwardly counting through all of the cells of the Canvas three times, performing the following operations on each cell (*i*, *j*) in turn:

1. Calculating the utility of cell (*i*, *j*)
2. Picking the best-performing strategy in the neighbourhood to occupy cell (*i*, *j*)
3. Fixing the pixel value of cell (*i*, *j*) to match that best-performing strategy

## 9.5 Calculation of Utilities

So first, the *utility* of any cell (after the next iteration of the IPD) is calculated as follows, making use of a *utilities* list which provides a sort of "look-up table" to reduce the need for calculation:

```
# Define utilities list based on payoffs
util = [0, 3*n, 3*n, 3*n, 0, 5*n, n+4, 0, 0, 0, 0, 0, n, 0, n-1]

# Function to calculate utility for each cell and store within colour code
def utility(x,y):
    this = pixcol(x,y) & 7
    utot = 0
    for i in range(-1,2):
        for j in range(-1,2):
            if (i!=0) or (j!=0):
                flag = this|(pixcol((x+width+i)%width,(y+height+j)%height))&15
                utot = utot + util[flag]
    pixset(x,y,utot*0x100 + (pixcol(x,y)&15)) # Utility is multiplied by 256
```

Here the variable *this* is first set to the value of the last three bits of the colour code (by bitwise *ANDing* with 7, which is 111 in binary). Then *utot* is set to zero, to initialise the utility count. Next, we count through the neighbourhood of 9 cells (with *i* and *j* both ranging over –1, 0, and +1), with the exception of cell (*x*, *y*) itself (where *i* and *j* are both 0). For each of these 8 neighbouring cells:

*flag* is set equal to *this*, bitwise *ORed* (the "|" operator ) with the last four bytes of the adjacent colour code. The upshot is that the value of *flag* will be set according to the following table:

| | | Neighbouring cell | |
|---|---|---|---|
| ***FLAG VALUE*** | blue = 1100 | green = 0010 | red = 0001 |
| This cell: blue = 100 | 1100 = 12 | 0110 = 6 | 0101 = 5 |
| This cell: green = 010 | 1110 = 14 | 0010 = 2 | 0011 = 3 |
| This cell: red = 001 | 1101 = 13 | 0011 = 3 | 0001 = 1 |

Within this table, the only duplication is between the red/green and green/red combinations, and since these yield exactly the same utilities either way round, that duplication doesn't matter at all. Now suppose that we ask: what should be the utilities accruing to the current cell, if (with the colour indicated at the left of this table) it was to be involved with a sequence of *n* interactions with an adjacent cell (with the colour at the top of this table)? This is what we should expect:

| | | Neighbouring cell | |
|---|---|---|---|
| ***EXPECTED UTILITY*** | blue = ALLD | green = TFT | red = ALLC |
| This cell: blue = ALLD | $n$ | $n + 4$ | $5 * n$ |
| This cell: green = TFT | $n - 1$ | $3 * n$ | $3 * n$ |
| This cell: red = ALLC | 0 | $3 * n$ | $3 * n$ |

Most of these values are straightforward, because ALLD versus ALLD scores 1 on every turn; ALLD versus ALLC scores 5 for ALLD and 0 for ALLC on every turn; any combination of TFT and ALLC scores 3 for each on every turn; so the only complication is ALLD versus TFT, where ALLD scores 5 and TFT 0 on the first turn, but both score 1 on each turn from then on, thus totalling $n$+4 for ALLD after $n$ turns, and $n$–1 for TFT.

And now we can see that this table matches exactly with the value that we get if we find the element of the *util* list with index value *flag*, as shown in the previous table:

```
util  =  [ 0, 3*n, 3*n, 3*n, 0, 5*n, n+4, 0, 0, 0, 0, 0, n, 0, n-1]
(index flag)  -   1    2    3   -   5    6   -  -  -  -  - 12 13  14
```

Thus the line:

```
utot = utot + util[flag]
```

will sum the total utility of the cell (*x*, *y*), as desired. Once this has been completed, the command:

```
pixset(x,y,utot*0x100 + (pixcol(x,y)&15)) # Utility is multiplied by 256
```

sets the pixel value of cell (*x*, *y*) to the total utility multiplied by 256 (hexadecimal #100), plus the last four bits of the existing pixel value (which are pulled out by *ANDing* that colour code with 15 (1111 in binary). So importantly, this new pixel value stores the new utility value *without affecting the last four bits*, which means that the processing of utility for neighbouring cells will not be affected. But note that this does affect the visible colour quite significantly, as you will see if you "comment out" (i.e. insert "#" before) the "noupdate()" command which prevents the Canvas from updating while all this is taking place. If the Canvas is allowed to update, the colours will change while the processing is going on, revealing how the results of all these intermediate calculations are indeed being stored in the pixel values.

## 9.6 Picking and Fixing the Best-Performing Neighbour

Here is the code for identifying the best-performing cell strategy within the neighbourhood of cell (*x*, *y*), including that cell itself:

```
# Function to pick the next occupant of (x,y)
def pickbest(x,y):
    bestsofar=pixcol(x,y)
    if randrange(5)>0: # With probability of 20%, retain current occupant
        for i in range(-1,2):
            for j in range(-1,2): # Otherwise, find best in neighbourhood
                if (pixcol((x+width+i)%width,(y+height+j)%height)&0xFFFF00)
                          >(bestsofar&0xFFFF00):
                    bestsofar=pixcol((x+width+i)%width,(y+height+j)%height)
    pixset(x,y,(pixcol(x,y)&0xFFFF0F) + (bestsofar & 15)*16)
```

This first makes *bestsofar* equal to the pixel colour of the current cell (*x*, *y*), and with probability 1 in 5, leaves that unchanged (so as to introduce irregularity into the cellular pattern, which can otherwise become artificially regular). Otherwise, it counts through every cell in the neighbourhood, with both *i* and *j* ranging from –1 to 1 inclusive,[10] and assesses whether that cell's utility measure is greater than the utility represented by *bestsofar*. It does that by *ANDing* both that cell's pixel value and *bestsofar* with the hexadecimal number #FFFF00, which yields the relevant utility value. If it is greater, then *bestsofar* is made equal to the new cell's pixel value, and the loop continues.[11] Once completed, the pixel at (*x*, *y*) is set to:

```
(pixcol(x,y) & 0xFFFF0F) + (bestsofar & 15) * 16
```

Here "`pixcol(x,y) & 0xFFFF0F`" sets the penultimate "nybble" (i.e. hexadecimal digit) of the pixel colour to 0, after which adding "`(bestsofar & 15) * 16`" *replaces* that penultimate nybble with the final nybble of *bestsofar* (which encodes the relevant strategy) multiplied by 16 (to move it "4 bits to the left"). The upshot of all this is that by the end of this process, the initial colour of cell (*x*, *y*) – which of course stores the count of its current utility (but not in its final byte) – is unaffected *except* in its penultimate nybble, and this has now been set to the four-bit code of the strategy with which it should be replaced in the next generation.

After all the cells of the Canvas have thus been processed, all that remains is to go through each cell in the Canvas, taking that penultimate nybble of its new colour code, and using it as the basis for choosing the colour which the cell should have in the next generation:

```
# Function to fix the colour
def fixbest(x,y):
    if (pixcol(x,y) & 16) > 0:  # if 16 bit is set, red
        pixset(x,y,0xFF0001)
    elif(pixcol(x,y) & 32) > 0: # if 32 bit is set, green
        pixset(x,y,0x00FF02)
    else:
        pixset(x,y,0x0000FC)  # otherwise, blue
```

Recall that the bottom four bits of the *red*, *green*, and *blue* cells are 0001, 0010, and 1100 respectively. Multiplication by 16 (as explained in the previous paragraph) shifted these 4 bits to the left, and thus from the last to the penultimate "nybble". Hence the *red* indicator will now be in the 16 bit and the *green* indicator in the 32 bit – if neither of these is set, then the new colour should be blue.

---

[10] Note the use of the *modulus* operator "%" in "pixcol((x+width+i)%width,(y+height+j)%height)" to ensure that the coordinates "wrap around" from the top to the bottom of the Canvas and from the left to the right.

[11] Note that *bestsofar* is replaced only if its *utility* value (after *ANDing* out the last four bits) is greater – this ensures that the colour of the cell (*x*, *y*) will change only if a neighbouring cell achieves a strictly higher utility than it does.